Using ESOOP in Squeak

Hesam Samimi

This material is based upon work supported in part by the National Science Foundation under Grant No. 0639876. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

VPRI Memo M-2009-012a

Viewpoints Research Institute, 1209 Grand Central Avenue, Glendale, CA 91201 t: (818) 332-3001 f: (818) 244-9761

Tutorial

Using ESOOP in Squeak
(Executable Specifications in Object Oriented Programs)

Hesam Samimi

last revised on: Oct, 2009

Contents

1 Introduction

This is a tutorial of a methodology to allow First Order Logic (FOL) specification methods for Squeak classes, and thus opening the possibility of using a SAT-based Constraint Solver to find models satisfying those specs.

Once the ESOOP change-set is loaded, and a unicode true-type font is installed as the code font, users can define methods and new classes in a fashion familiar to Squeak programmers. Given a running Solver server, objects can use the *solve* directive to satisfy their specifications. Two step by step examples, Sorting a List of integers, and modeling the Pentominoes puzzle game are presented in this tutorial.

2 Installation

2.1 Install Unicode Font (* required only if logical symbols instead of ASCII desired):

The specifications allow for unicode symbols for logic. However, the english spellings of them, including *and*
, *or*, *such that*, *in*, *prime* are also supported. This step is optional.

a. download DejaVu fonts:
   http://prdownloads.sourceforge.net/dejavu/dejavu-fonts-ttf-2.29.tar.bz2?download
b. drag unicode font *DejaVuSans.ttf* onto the *Moshi* image
c. *left click → appearance → system fonts → code font → DejaVuSans → 12 pt*

You may run the code below in Squeak's workspace in order to copy and paste the unicode characters used in logic syntax:

```
String streamContents: [:s | 16r2032 to: 16r22FF do:
  [:c | s nextPut: c asCharacter. c \\ 16 = 15 ifTrue: [s cr]]].
```

2.2 Load the ESOOP change-set

a. download the change-set:
   http://www.cs.ucla.edu/~hesam/b-brains/runnable-specs/squeak/ESOOPLATEST.cs
b. drag the downloaded change-set into the *Moshi* image, select *filein entire file.*

3 Bringing up the solver server

The solver is installed on *tinlizzie.org* machine. Before using the *solve* directive, ensure the *Kodkod* server is up and running on this machine, by executing the following script. Note that in the current setup, the server quits after being called once, so this step must be repeated after each call to solve. Do this at steps 4.2, and 6.2.

```
you@tinlizzie:~$ /home/hesam/tools/solver/Kodkod/run-kodkodi-server.sh
Kodkod server listening for clients on port 9128...
```

The server options are contained in *setOpts* method of the *ESOOPLogSolver* class.

4 Defining spec methods for an existing Squeak class

In this section, we will define a *sort* goal for an existing classes, an *OrderedCollection* as well as *Point*. In order to not mock with existing Squeak classes, we have defined a subclass of OrderedCollection, *ESOOPList,* to do this experiment. Similarly, there is a *ESOOPInteger* class for *Integer primitive.*

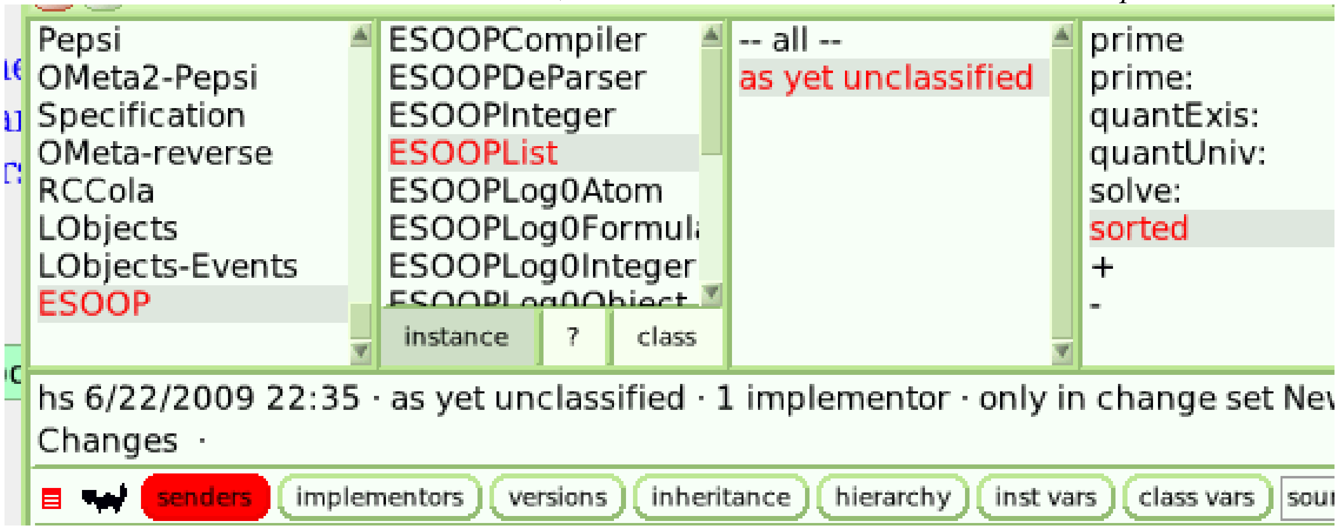
Unlike Squeak, the name of the class is included before the method name. Note that in the method definition below, the prefix ESOOP is not included within the code; It's added automatically. The usual logical symbols are used in the specifications. However, their English spellings are also supported.

The *prime* message (or symbol ') subsequent to a variable denotes its value in the solution. When prime version of a variable exists in the formula, the solver should treat the variable as an unknown, and allowed to explore values for it other than its current value. On the other hand, when no formula mentions the variable prime, this is directing the solver to consider the variable fixed.

The directive *predicate* starts the syntax for method definition. This will distinguish between these methods and those representing *goals.* Although there are currently no differences, between predicate and goal methods, one might potentially want to do extra work for goal methods.

4.1 Defining sort specs for List

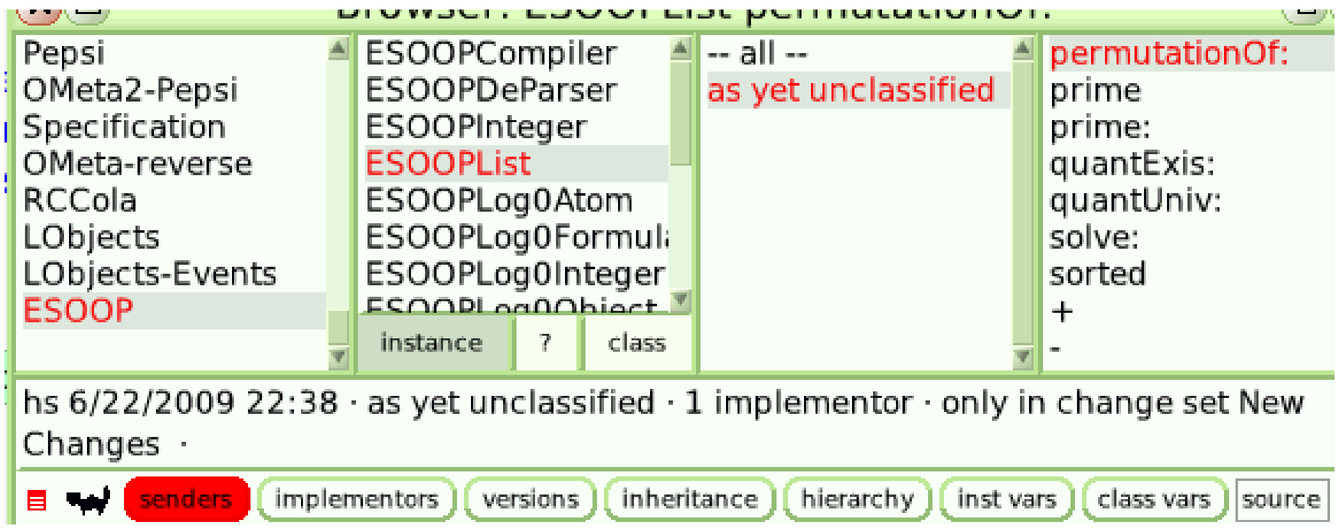
a. define a new method for *ESOOPList,* named *sorted*. Use the usual *cmd-s* to *accept*.



hs 6/22/2009 22:35 · as yet unclassified · 1 implementor · only in change set New Changes ·

```
predicate List sorted
    all i in self indices allButLast | (self at: i) <= (self at: (i+1)).
```

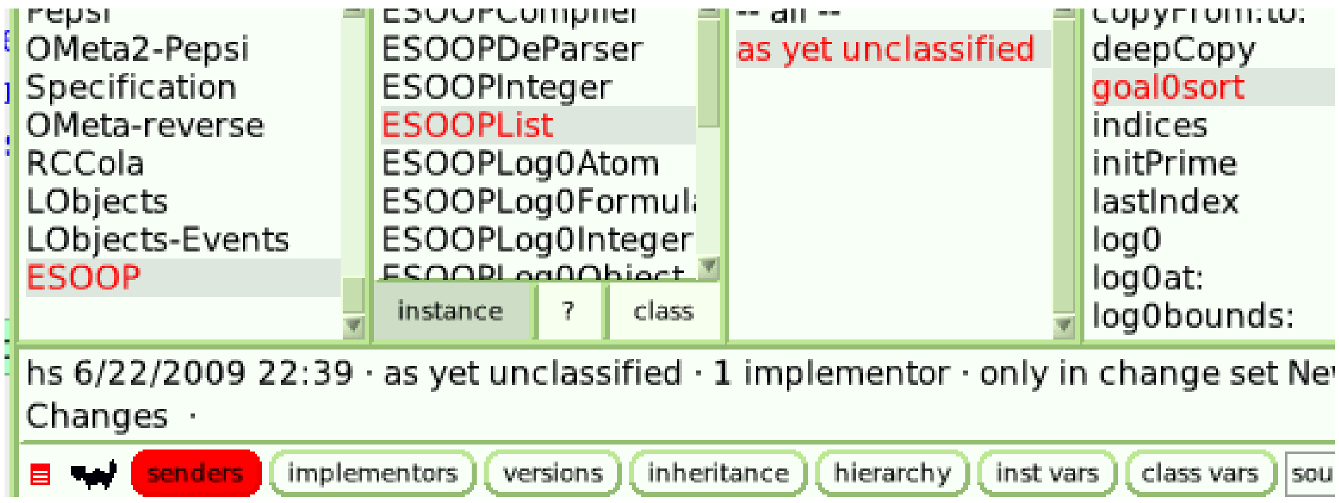
b. define a new method *permutationOf:* for the same class.



hs 6/22/2009 22:38 · as yet unclassified · 1 implementor · only in change set New Changes ·

```
predicate List permutationOf: L
    all e in all Integer | self occurrencesOf: e = L occurrencesOf: e.
```

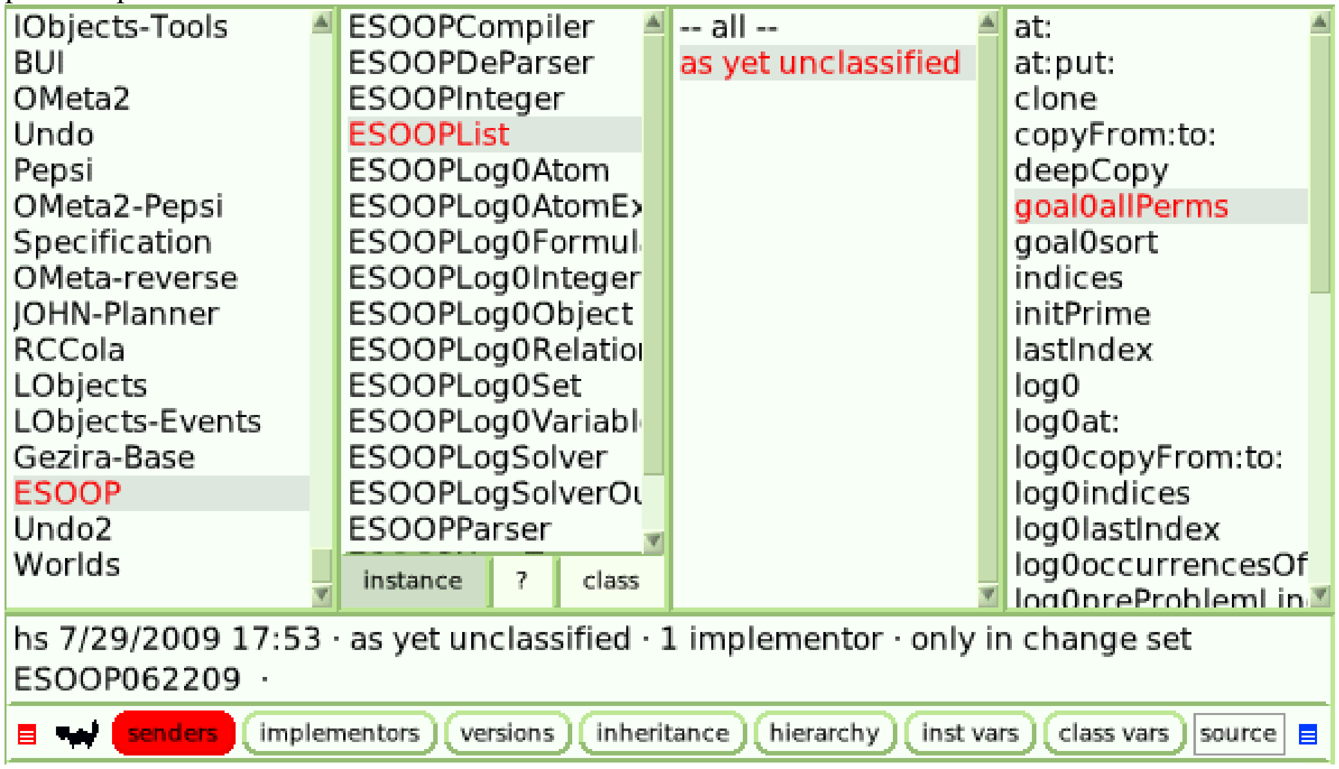
c. define a goal method *sort* for the *ESOOPList* class. This method will be saved as *goal0sort.*



hs 6/22/2009 22:39 · as yet unclassified · 1 implementor · only in change set Ne
Changes ·

```
goal List sort
    self prime permutationOf: self and
    self prime sorted.
```

d. let's also define a goal method *allPerms* for the *ESOOPList* class. We will later use it to get all possible permutation of a list.



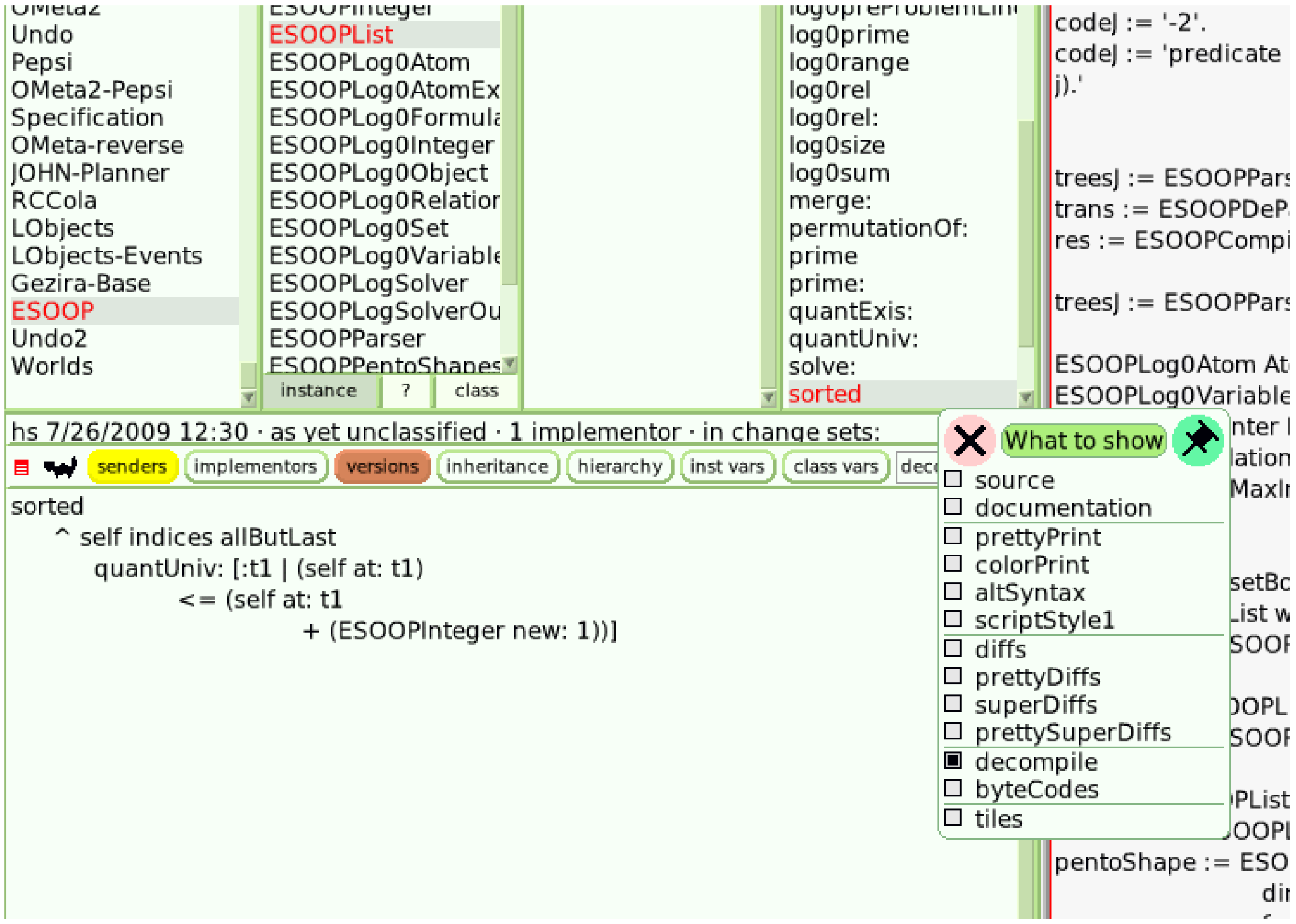
hs 7/29/2009 17:53 · as yet unclassified · 1 implementor · only in change set
ESOOP062209 ·

```
goal List allPerms
    self prime permutationOf: self.
```

4.2 Viewing and running de-sugared predicates

The defined specification predicates are simply syntactic sugar for Squeak methods, and it is possible to call them just as an ordinary Squeak method.

a. To see the pure Squeak version of the defined List methods sorted and pemutationOf: methods, select *decompile* instead of *source* in the *What to show* menu shown below:



```
sorted
    ^ self indices allButLast
        quantUniv: [:t1 | (self at: t1)
                <= (self at: t1
                        + (ESOOPInteger new: 1))]
```

b. This predicate can be run simply by instantiating an *ESOOPList* object and calling *sorted*. The previously defined goal predicate *sort* can also be called with its compiled name: *goal0sort.* Running a goal method simply returns a boolean stating whether or not the goal predicate evaluates to true in the current state of the object.

```
l1 := ESOOPList withAll: (#(3 4 5 1 2) collect: [:x | ESOOPInteger new: x ]).
l1 sorted.    "  returns false "
l1 goal0sort. " returns false "
```

4.3 *solve:* directive: Sorting a List using sort specification

a. bring up the Kodkod server using the command stated in section 3.

b. run the following test. Note that Integer type needs to be bounded first, and Integers must be of specialized type: ESOOPInteger.

```
testSort

   "
     * bring up solver server first *
     ESOOPCompiler testSort.
   "
   | l1 |

   ESOOPInteger setBounds: #(0 30).
   l1 := ESOOPList withAll: (#(3 4 5 4 2 1) collect: [:x | ESOOPInteger new: x ]).
   l1 solve: #sort.
   ^l1
```

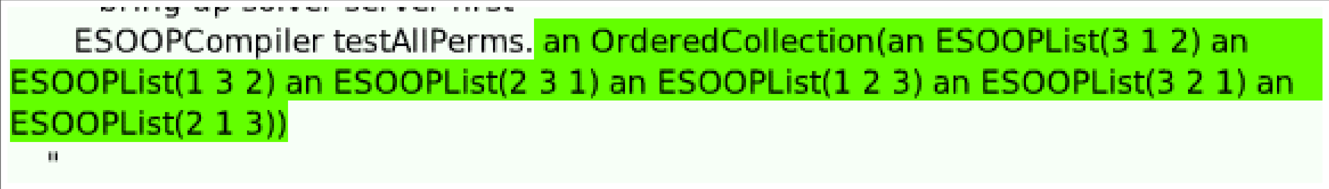
4.4 *solveAll:* directive: Permutations of a List using specification

```
testAllPerms

   "
     STEP 3: Run test (after previous steps)
     * bring up solver server first *
     ESOOPCompiler testAllPerms.
   "
   | l1 |

   ESOOPInteger setBounds: #(0 30).
   l1 := ESOOPList withAll: (#(1 2 3) collect: [:x | ESOOPInteger new: x ]).
   ^l1 solveAll: #allPerms.
```

Result of *l1 solveAll: #allPerms*



```
     ESOOPCompiler testAllPerms. an OrderedCollection(an ESOOPList(3 1 2) an
ESOOPList(1 3 2) an ESOOPList(2 3 1) an ESOOPList(1 2 3) an ESOOPList(3 2 1) an
ESOOPList(2 1 3))
   "
```

4.5 Defining specs for Point class (used for later case study)

ESOOP supports enabling executable specifications for existing Squeak classes. This is possible by defining a new specialized subclass of the class in question, along with listing particular instance variables of which are involved in the specifications.

Lets use the *Point* class as an example. In section ? we will present a PentominoShapes example as case study which finds the possible Pentomino puzzle pieces using a solver. The class holds several instance variables as collection of Points. Since the problem involves Point class, we need to first define an *ESOOPPoint* class, as subtype of Point which can execuate specifications. This is accomplished via the following method call.

Syntax for instance variable type specification:

```
Scalar:        {<name>. <type>. <modifiable?>}
Collection:   {<name>. <type>. <modifiable?>. <list element type>. <list size>}
```

Syntax for defining a subtype of an existing class, supporting executable specifications:

```
ESOOPUserTypes class: <class> subclass: <subclass>
      instanceVariables: {<instVar1 type>.
                          <instVar2 type>
                          … }
```

Note that the *ESOOP* prefix for the *ESOOPInteger* and *ESOOPList* types is not included in the type names. The prefix is also automatically added to the name of subclass.

a. define subtype of Point called ESOOPPoint by evaluating the following code

```
ESOOPUserTypes class: #Point subclass: #Point
     instanceVariables: {{#x. #Integer. true}.
                         {#y. #Integer. true}}
```

b. define *equals:* method for ESOOPPoint. The *prime* refers to the desired value in the solution:

```
predicate Point equals: p
   self x prime = p x prime and
   self y prime = p y prime.
```

c. define *plus:equals:* method for ESOOPPoint:

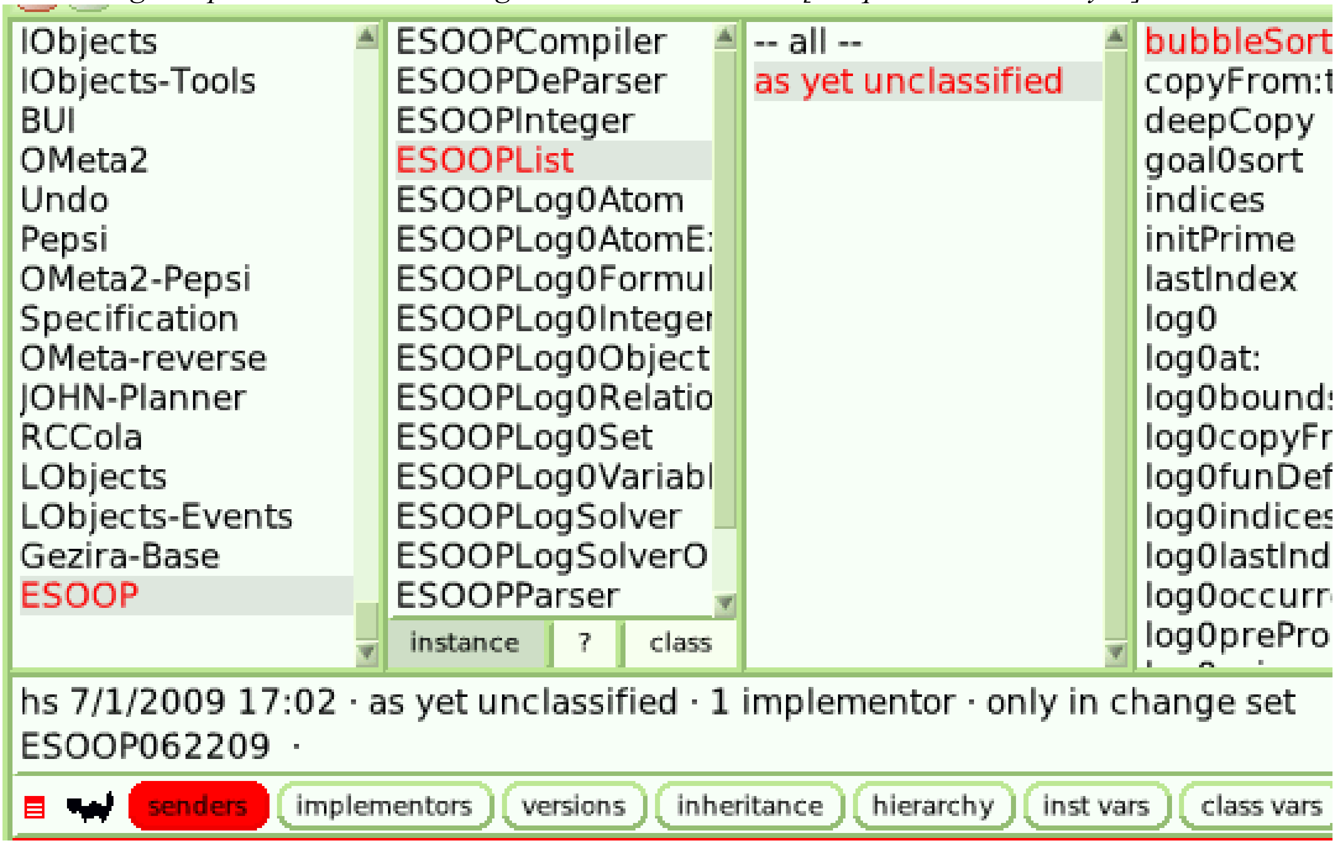
```
predicate Point plus: p equals: q
   self x prime + p x prime = q x prime and
   self y prime + p y prime = q y prime.
```

5 Defining Squeak methods as optimizations for specs

Relying on executable specifications is rarely practical due to slowness. In practice, we would like to write implementations for the task. When implementations fail to work, either giving incorrect results, or resulting in errors, we want to use an automatic fallback mechanism to rely on the executable specifications to do the job. This fallback feature is supported as follows.

Write bubbleSort for List class, as an obtimization for the sort goal

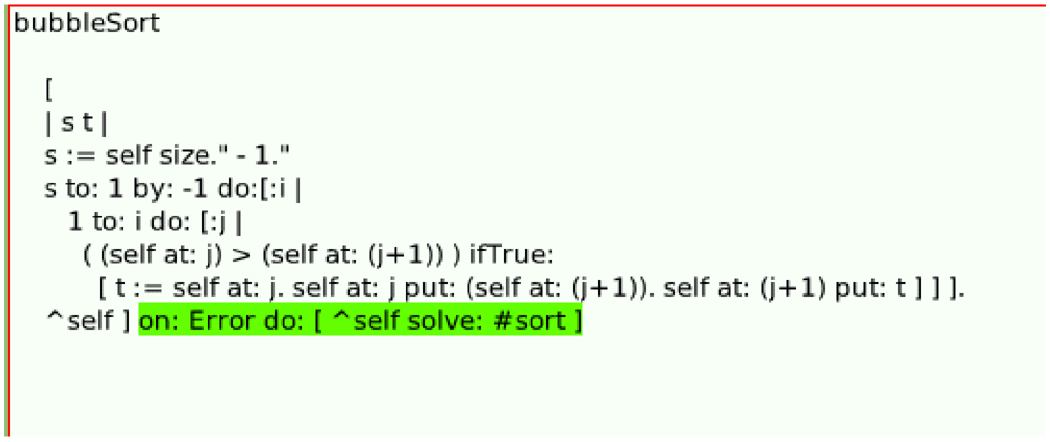
a. Syntax for a squeak method as a goal optimization is:
goalOptimization <class> <goal> <method name> [<squeak method body>].



hs 7/1/2009 17:02 · as yet unclassified · 1 implementor · only in change set ESOOP062209 ·

```
goalOptimization List sort bubbleSort
     [  | s t |
      s := self size - 1.
      s to: 1 by: -1 do:[:i |
        1 to: i do: [:j |
          ( (self at: j) > (self at: (j+1)) ) ifTrue:
            [ t := self at: j.
              self at: j put: (self at: (j+1)).
              self at: (j+1) put: t ] ] ].
     ^self ].
```

Internally, this defines a method *bubbleSort* for class *ESOOPList* that looks like below.



```
bubbleSort

  [
  | s t |
  s := self size." - 1."
  s to: 1 by: -1 do:[:i |
    1 to: i do: [:j |
     ( (self at: j) > (self at: (j+1)) ) ifTrue:
       [ t := self at: j. self at: j put: (self at: (j+1)). self at: (j+1) put: t ] ] ].
  ^self ] on: Error do: [ ^self solve: #sort ]
```

When implementation methods work as expected, specifications are not used. However,the fallback mechanism will kick-in if the implementation results in error. For example, in the shown *bubbleSort* method, if the "– *1"* is uncommented, the method will result in an out of bounds array indexing error. In such a case, as shown above, the object will use the *solve:* directive to accomplish its sorting goal, relying on the specification of *sort* and the constraint solver.

Currently the fallback mechanism does not kick in if the implementation method results in incorrect values, yet without causing an error. This can be easily added by checking the goal predicate, in this case *sort* for the object in question. When the implementation does not satisfy the goal, the fallback should occur.

6 Defining a new class supporting executable specs

In this section, we want to model the Pentominoes puzzle pieces, relying on the declarative specifications to find a possible shape and orientation of a valid pentomino piece. Let's model the problem by defining a class *PentominoShapes*, with instance variables *dirs, squares, froms,* and *fromDirs*, enough to specify the problem. The *dirs* is a fixed collection of 4 points (0@1 0@-1 -1@0 1@0) representing the possible four directions up, down, left, and right. Note that we mark this instance variable as non-modifiable. Note that this problem involves Squeak's existing class, *Point*, hence we created an *ESOOPPoint* subclass in previous sections. The *sqaures* variable will be a list of 5 points representing the coordinates of squares that make the pentomino piece, and the value in the solution will be an answer to the problem. The *froms* is also a 5 element list, describing how each square is obtained from a previously indexed square. The values in this collection gives the index of square each square is derived from by adding the origin square, to one of the direction points, *dirs*. The *fromDirs* is there to tell which direction is chosen for each of the squares.

All new user defined classes that involve executable specifications will be subclasses of a class named *ESOOPUserTypes.* This class allows the syntax of these specification methods to be of FOL as examples above, and provides the method *solve*.

Defining a new class is done via passing the message *subclass:instanceVariables:* to the *ESOOPUserTypes* class. The main difference compared to normal Squeak is that instance variables are typed. Also, each instance variable type specifications includes a boolean value stating whether or not the value should be modifiable or not. This is necessary to know which variables is the solver allowed to modify in order to solve a problem. For *List* type variables, the type of elements and the size is also specified. The format was given in section 4.3. Once again note that the *ESOOP* prefix for the *ESOOPInteger* and *ESOOPList* types is not included in the type names. Also, this method of class definition automatically defines getters and setters for all instance variables.

6.1 Defining a new class

A *subclass:instanceVariables:* call to *ESOOPUserTypes* class defines a new class. This method along with the sample call below is found as a class method for *ESOOPUserType* in the browser.

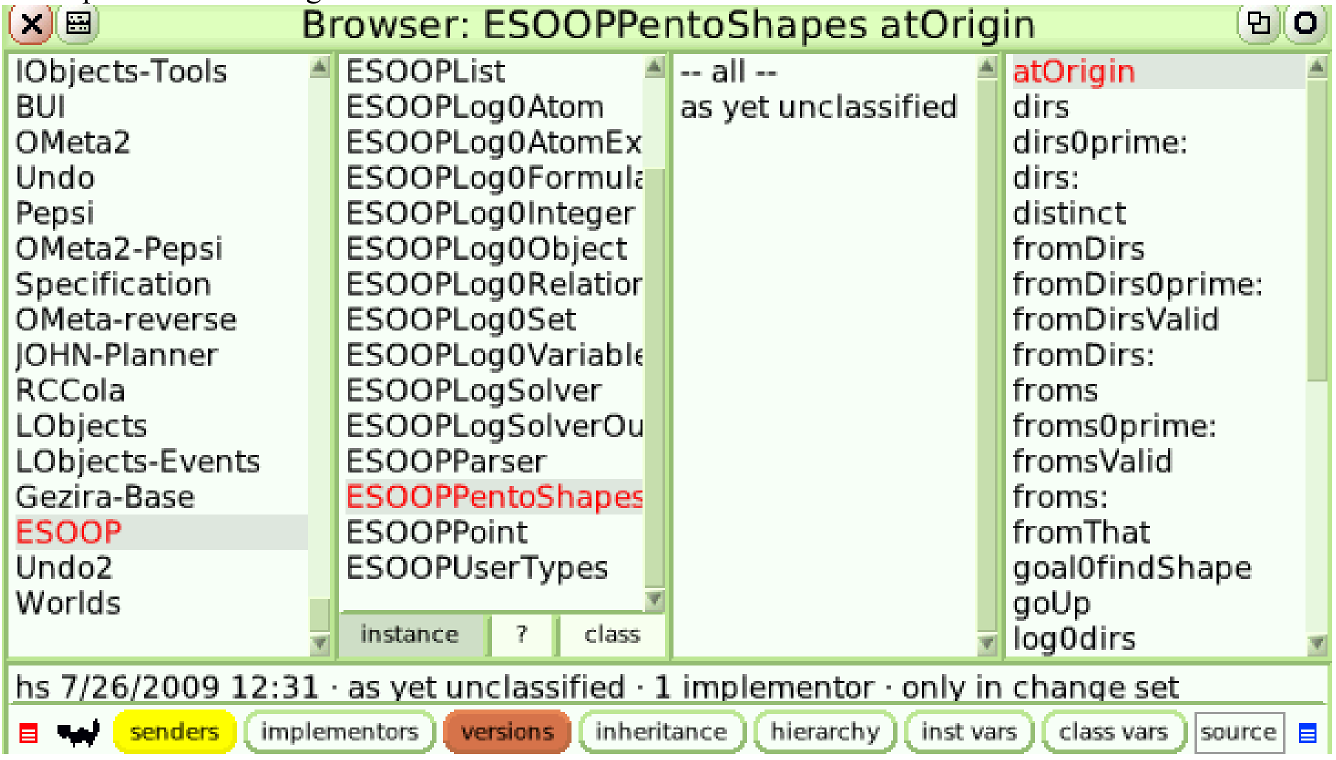
```
ESOOPUserTypes subclass: #PentoShapes
  instanceVariables: {{#dirs. #List. false. #Point. 4}.
                      {#squares. #List. true. #Point. 5}.
                      {#froms. #List. true. #Integer. 5}.
                      {#fromDirs. #List. true. #Integer. 5}}.
```

7 Defining spec methods for the new class

We added the *PentoShapes* class in the previous section, which was saved in the browser as *ESOOPPentoShapes* in order to keep these separate from Squeak's own classes. Now let's specify the problem of finding a valid shape and orientation for a pentomino piece.

7.1 Defining *findShape* specs for *PentoShapes*

a. define a new method for *PentoShapes,* named *atOrigin.* The predicate specifies that we want the first square to be the origin.



```
predicate PentoShapes atOrigin

   (self squares at: 1) x prime = 0 and
   (self squares at: 1) y prime = 0.
```

b. define a new method for *PentoShapes,* named *fromsValid.* The predicate states that froms variable has valid index values.

```
predicate PentoShapes fromsValid

   all i in self froms prime allButFirst | i > 0.
```

c. define a new method for *PentoShapes,* named *fromDirsValid.* The predicate specifies that has valid index values.

```
predicate PentoShapes fromDirsValid

   all i in self fromDirs prime | i > 0 and i <= 4.
```

d. define a new method for *PentoShapes,* named *goUp.* The predicate specifies that all squares should be on y-positive quadrants.

```
predicate PentoShapes goUp

   all s in self squares | s y prime >= 0.
```

e. define a new method for *PentoShapes,* named *noLeftFirstRow.* The predicate avoid redundant shapes disallowing negative x-value squares for y=0 row.

```
predicate PentoShapes noLeftFirstRow

   all s in self squares | s y prime > 0 or s x prime >= 0.
```

f. define a new method for *PentoShapes,* named *fromThat.* The predicate ensures each square is derived via a previously indexed square, excluding the first square at origin.

```
predicate PentoShapes fromThat

   all i in self froms prime indices allButFirst | (self froms prime at: i) < i.
```

g. define a new method for *PentoShapes,* named *distinct.* The predicate ensures square coordinates are unique.

```
predicate PentoShapes distinct

   all s1, s2 in self squares | s1 = s2 or (s1 equals: s2) not.
```

h. define a new method for *PentoShapes,* named *sprouted.* The predicate specifies that each subsequent square is derived via a previously index square plus one of the direction points.

```
predicate PentoShapes sprouted

   all i in self squares indices allButFirst |
 (self squares at: (self froms prime at: i)) plus:
    (self dirs at: (self fromDirs prime at: i)) equals:
        (self squares at: i).
```

i. finally define the *findShape* goal for the class, specifying the above predicates must hold:

```
goal PentoShapes findShape

   self fromDirsValid and
   self fromsValid and
   self atOrigin and
   self fromThat and
   self goUp and
   self noLeftFirstRow and
   self distinct and
   self sprouted.
```

We're now ready to run the specifications above. It should be noted that the instance variables with primitive values should currently use the ESOOP specific subclass, namely *ESOOPInteger* and *ESOOPList.*

7.2 Executing *PentoShapes findShape* goal specification

a. bring up the Kodkod server using the command stated in section 3.
b. run the following test (also found under ESOOPCompile → testPentominos)

```
testPentominos

   "
     * bring up solver server first *
     ESOOPCompiler testPentominos.
   "


   | dirs squares fromDirs froms pentoShape |

ESOOPInteger setBounds: #(-3 5).
dirs := ESOOPList withAll: (#((0 1) (0 -1) (-1 0) (1 0)) collect:
            [:p | ESOOPPoint new x: (ESOOPInteger new: p first);
                                    y: (ESOOPInteger new: p second); yourself ]).
squares := ESOOPList withAll: ((1 to: 5) collect:
            [:p | ESOOPPoint new x: (ESOOPInteger new: 0);
                                    y: (ESOOPInteger new: 0); yourself ]).
froms := ESOOPList new.
fromDirs := ESOOPList new.
pentoShape := ESOOPPentoShapes new
                     dirs: dirs; squares: squares; fromDirs: fromDirs;
                     froms: froms; yourself.
pentoShape solve: #findShape.
^pentoShape squares.
```

c. note that after the *solve: #findShape* call, the squares instance variable's points are updated to reprent a valid Pentomo shape.

6.2.a (before)	6.2.b (after)
an ESOOPList(0@0 0@0 0@0 0@0 0@0)	an ESOOPList(0@0 1@0 0@1 2@0 -1@1)

8 Current limitations and known bugs

In the current implementation, Squeak's own primitive and library types are not tampered for safety, and their subclasses clearly marked as *ESOOP<type>* such as *ESOOPList* and *ESOOPInteger* are used.

Another limitation is that instance variables can only take values of supported types: *ESOOPInteger*, *ESOOPList*, or user defined classes *ESOOP<type>*. The only reason is that objects need to support the *prime* message (post solution value), and several methods regarding the logical expression equivalent of objects. When these are defined within Squeak's own primitives and library types, no such limitation exists. Currently there is a regrettable need to specify at the time of class definition, whether each instance variable is modifiable or not. This in theory isn't needed, since the "primes" in the problem specification should tell us this information.

Also, the List sizes need to be declared in the types, which is limiting.
A main limitation of the logic solver we use is bounded model checking and efficiency. The solver requires everything involved in the problem including integers to be bounded. It may not realistic to bound integers to some low interval, depending on the problem. Very large bounds would increase the run-time in some cases prohibitively.

Another limitation of executable specification methods is that they may not be recursive. These methods are only used by in-lining the expressions to make a final string representing the problem in the syntax of the logic solver. This problem needs to be addressed.

A major bug in current implementation is due to usage of class variables such as counters, etc. which make the code flaky. Sometimes multiple tries are required until a *solve:* call works properly.

Currently there is a bug regarding method definitions. When defining an ESOOP method, few extra methods (logical counterparts of the OOP method) are automatically defined. While all four methods automatically generated out the the user typed method get added to the class dictionary, only the two main ones (named *<method name>* and *log0<method name>*) are shown in the browser. The other helper methods do not show up in the browser. The user needs to search and delete them manually (maybe via the class dictionary) when the methods are no longer needed.

Although the *Kodkod* solver does allow using cost optimization SAT solvers, we have not experimented with such a tool. Cost optimization problems, such as the second example above, are very common and need to be handled properly.

In the current setup, the solver server can only handle one connection at a time, and quits with after the first call. It's possible to modify the code to keep the server running.

Sometimes when calling upon the solver, the program stalls, likely to be a parsing/communication issue between the program and the server. Aborting the solver server and retrying, usually resolves the problem.

As for future work, we are going to have to work on formulations to incorporate events in also a declarative way, as well as support for introduction of optimizations to the specifications.